**POINTERS**

* Char \*ptr=”geeksforgeeks”

The operator \* is used for dereferencing and the operator & is used to get the address. These operators cancel out effect of each other when used one after another. We can apply them alternatively any no. of times. In the above code, ptr is a pointer to first character of string ‘g’. \*ptr gives us g, &\*ptr gives address of g, \*&\*ptr again g, &\*&ptr address of g, and finally \*&\*&ptr gives ‘g’

* Sizeof should never be used for array as a parameter

Example:- void fun(int a[] or int \*a)

{

Int n=sizeof(a)/sizeof(a[0]);

If we try printing array with n…complete array may not be printed because only address of first element is passed..not complete array and we access that array through base address

}

* #include<stdio.h>
* int main()
* {
* int arr[] = {10, 20, 30, 40, 50, 60};
* int \*ptr1 = arr;
* int \*ptr2 = arr + 5;
* printf("Number of elements between two pointer are: %d.",
* (ptr2 - ptr1));
* printf("Number of bytes between two pointers are: %d",
* (char\*)ptr2 - (char\*) ptr1);
* return 0;
* }

Output:5,20

 When we do ‘(char \*)ptr2′, ptr2 is type-casted to char pointer and size of character is one byte, pointer arithmetic happens considering character pointers. So we get 5\*sizeof(int)/sizeof(char) as a difference of two pointers.

void swap(int \*px, int \*py)

{

   \*px = \*px - \*py;

   \*py = \*px + \*py;

   \*px = \*py - \*px;

}

>> May generate segmentation fault if value at pointers px or py is constant or px or py points to a memory location that is invalid  
>> May not work for all inputs as arithmetic overflow can occur

* Int a=5;

Void \*ptr=(int\*)&a;

Printf(“%d”,\*ptr); //compile time error..we need to type cast ptr as….printf(“%d”,\*(int\*)ptr)

* To swap two strings:-

1. If we use pointer to store strings then we should use double pointer to swap the strings;
2. If we use character array to store strings then we should use data copy method to swap the strings;
3. See the program in uca folder

#include <stdio.h>

void f(char\*\*);

int main()

{

    char \*argv[] = { "ab", "cd", "ef", "gh", "ij", "kl" };

    f(argv);

    return 0;

}

void f(char \*\*p)

{

    char \*t;

    t = (p += sizeof(int))[-1];

    printf("%s\n", t);

}

Output:gh

>> char \*pChar;

int \*pInt;

float \*pFloat;

sizeof(pChar);

sizeof(pInt);

sizeof(pFloat);

Assume that int,float have 4 bytes and char 1 bytes..pointer size is also 4 bytes. Output:4,4,4

Irrespective of the type of pointer, the size for a pointer is always same. So whether it’s pointer to char or pointer to float, the size of any pointer would be same. Even size of a pointer to user defined data type (e.g. struct) is also would be same.

>>

i) sizeof(int)

ii) sizeof(int\*);

iii) sizeof(int\*\*); assume that size of pointer if 4 bytes and size of int is also 4 bytes

ouput:size of each is 4

Size of all pointer types is same.

And whether it is a ‘pointer to char’ or ‘pointer to int’ or ‘pointer to pointer to int’, the size always remain same.

That’s why all i), ii) and iii) would compile successfully and would result in same size value of 4.

>> int var;

void \*ptr=&var;

\*ptr=5;

Printf(“%d”,\*ptr)// it will give compile time error

It should be noted that dereferencing of void pointer isn’t allowed because void is an incomplete data type. The correct way to assign value of 5 would be first to typecast void pointer and then use it. So instead of \*ptr, one should use \*(int \*)ptr.

>>

void mystery(int \*ptra, int \*ptrb)

{

   int \*temp;

   temp = ptrb;

   ptrb = ptra;

   ptra = temp;

}

int main()

{

    int a=2016, b=0, c=4, d=42;

printf("%d\t%d", a,b);

mystery(&a, &b);

    printf("%d\t%d", a,b);

}

here..swapping doesn’t occurs..only address are changed in the function which is local to function and changes are not reflected

Output: value of a and b before and after swaping remains same

->if we use function then swapping occurs:

void mystery(int \*ptra, int \*ptrb)

{

   int temp;

   temp = \*ptrb;

   \*ptrb = \*ptra;

   \*ptra = temp;

}

>>int (\* f)(int \*):a pointer to a function that takes

Integer pointer as an argument and returns and integer

>> **const char \*ptr|| char const \*ptr is same**: value at location of which ptr is holding the address cant be changed but ptr can be assigned with new address

#include <stdio.h>

int main(void)

{

    int i = 10;

    int j = 20;

    const int \*ptr = &i;    /\* ptr is pointer to constant \*/

    printf("ptr: %d\n", \*ptr);

    \*ptr = 100;        /\* error: object pointed cannot be modified

                     using the pointer ptr \*/

    ptr = &j;          /\* valid \*/

    printf("ptr: %d\n", \*ptr);

    return 0;

}

>> **char \*const ptr:** address ptr is holding cant be changed with new address

But value at location of which ptr is holding the address can be changed

#include <stdio.h>

int main(void)

{

   int i = 10;

   int j = 20;

   int \*const ptr = &i;    /\* constant pointer to integer \*/

   printf("ptr: %d\n", \*ptr);

   \*ptr = 100;    /\* valid \*/

   printf("ptr: %d\n", \*ptr);

   ptr = &j;        /\* error \*/

   return 0;

}

>>**const char \*const ptr**: neither address ptr is holding nor value at location of which ptr is holding address can be changed

#include <stdio.h>

int main(void)

{

    int i = 10;

    int j = 20;

    const int \*const ptr = &i;        /\* constant pointer to constant integer \*/

    printf("ptr: %d\n", \*ptr);

    ptr = &j;            /\* error \*/

    \*ptr = 100;        /\* error \*/

    return 0;

}

>>**Difference of “void\*” in c and c++?**

c allows void pointer to be assigned to any pointer type without cast whereas c++ doesn’t,Example:-

int \*p=malloc(sizeof(int));/\*malloc returns void pointer and is implicitly type casted in c\*/

>>**For c++:-**

int \*p=(int\*)malloc(sizeof(int));

**#Wild pointers:-**

Uninitialized pointers are wild pointers because they point to some arbitrary memory location and may cause program crash.

int main()

{ int \*p; //wild pointer

Int a=12;

P=&a; //now p is not a wild pointer}

>> int main()

{ int \*p=malloc(sizeof(int));

\*p=12; //valid ,assuming that malloc doesn’t returns NULL

}

**#Dangling pointers:-**

Pointer pointing to the memory location that has been deleted(or freed) is called Dangling pointer.(pointer pointing to invalid memory address i.e address of unreserved memory)

>>There are 3 different ways a pointer can act as dangling:-

1. **de-allocating memory :-**

int main()

{ int \*ptr=malloc(sizeof(int));

Free(ptr);

//now ptr is dangling as ptr still contains the address value of memory reserved which has now been freed.

Ptr=NULL;// now ptr is not dangling

}

1. **function call:-**

int \*fun()

{ int x=5;

Return &x;

}

int main()

{int \*p=fun();

//p is a dangling pointer because x is a local variable whose scope gets over after function is executed and memory for x is released but p still contains address of x.

Printf(“%d”,p); //garbage address

}

>>but if we make “x” as a static variable then scope of x will be through out the program,and then “p” will not be dangling pointer.

1. **Variable goes out of scope:-**

int main()

{ int \*p;

{ int x=5;

P=&x;

}

//p is a dangling pointer at this point because it contains the address of variable which is no longer exists.

P=NULL;//p is not dangling now because p doesn’t points any invalid memory;

}

**Important Points**

1. **NULL vs Uninitialized pointer –**An uninitialized pointer stores an undefined value. A null pointer stores a defined value, but one that is defined by the environment to not be a valid address for any member or object.
2. **NULL vs Void Pointer** – Null pointer is a value, while void pointer is a type

>> int \*ptr=NULL; //null pointer

>>**void pointers:-**

1. void pointers **cannot be dereferenced**. It can however be done using typecasting the void pointer
2. Pointer arithmetic is not possible on pointers of void due to lack of concrete value and thus size

>> In C, array parameters are always treated as pointers. So following two statements have the same meaning.

void fun(int arr[])

void fun(int \*arr)

[] is used to make it clear that the function expects an array, it doesn’t change anything though. People use it only for readability so that the reader is clear about the intended parameter type. The bottom line is, sizeof should never be used for array parameters, a separate parameter for array size (or length) should be passed to fun(). So, **in the given program, arr\_size contains ration of pointer size and integer size, this ration= is compiler dependent.**

#include <stdio.h>

void fun(int arr[], size\_t arr\_size)

{

  int i;

  for (i = 0; i < arr\_size; i++)

      printf("%d ", arr[i]);

}

int main()

{

  int i;

  int arr[] = {10, 20 ,30, 40};

  // Use of sizeof is fine here

  size\_t n = sizeof(arr)/sizeof(arr[0]);

  fun(arr, n);

return 0;

}

>> **int ( \* f) (int \* );** what does statement means?

**Explanation:** The steps to read complicated declarations :

1)Convert C declaration to postfix format and read from left to right.

2)To convert expression to postfix, start from innermost parenthesis, If innermost parenthesis is not present then start from declarations name and go right first. When first ending parenthesis encounters then go left. Once the whole parenthesis is parsed then come out from parenthesis.

3)Continue until complete declaration has been parsed.

At First, we convert the following given declaration into postfix:

int ( \* f) (int \* )

Since there is no innermost bracket, so first we take declaration name f, so print “f” and then go to the right, since there is nothing to parse, so go to the left. There is \* at the left side, so print “\*”.Come out of parenthesis. Hence postfix notation of given declaration can be written as follows:

f \* (int \* ) int

Meaning: f is a pointer to function (which takes one argument of int pointer type) returning int .

* ‘this’ pointer:

**The ‘this’ pointer is passed as a hidden argument to all nonstatic member function calls and is available as a local variable within the body of all nonstatic functions.** ‘this’ pointer is a constant pointer that holds the memory address of the current object. ‘this’ pointer is not available in static member functions as static member functions can be called without any object (with class name).

Class sample

{ private:

Int a;

Public:

Void set(int num)

{ a=num;}

};

Int main()

{

Sample s;

Int num=10;

s.set(num);

}

Explanation: when the function set() is called,the invoking object is ‘s’. Therefore the pointer ‘this’ will be automatically set to the address of ‘s’ and is implicitly passed as an argument to the member function.

>> s.set(num); is same as s.set(&s,num);

>> a=num is same as ( this->a=num or (\*this).a=num )

>> **return \*this** = used to return the current object from member function.